MyDAO Protocol Incentive Mechanism Design

January 12, 2023[1]

Description

This is documentation that details the purpose, design, results from, and analysis of the different incentive mechanisms that are employed in the MyDAO protocol network.

[1]Image credit: https://www.flaticon.com/authors/steven-edward-simanjuntak

1 Project Description

The MyDAO protocol enables its users to unlock dead capital by tokenizing electronic assets that they own. In exchange, these users (also called agents in the network), receive minted TOKN tokens whose quantity matches the dollar value of the tokenized asset. These tokens can be exchanged for ALGO or other assets on the network, effectively unlocking the value of the tokenized asset.

2 Why do we design incentives?

In a decentralized autonomous organization, there is no central governing entity to reprimand members whose behaviors are not in line with the goals of the organization, or to reward those who act accordingly. Instead, incentives are created to motivate these members to act is such ways that the goals of the organization are met.

We want the agents in the economy to act such that

1. good quality assets are tokenized
2. they assure these assets/ demonstrate that they are functional and thus have value
3. provide liquidity for the TOKN token

We want these behaviours because

1. asset tokenization will increase protocol revenue
2. assuring the functionality and therefore value of the assets builds confidence in the TOKN token which is backed by them on a 1:1 ratio with the dollar
3. when there is much liquidity, the price of TOKN is not volatile and therefore people are more confident in it. It is also readily exchangeable at a stable rate when there is sufficient liquidity. Combined, these mean that people find more utility in the token and are therefore more likely to use it. This confidence in and utility of the product is clearly desired.

3 Our incentive mechanisms

3.1 Incentives for asset tokenization

When one tokenizes their valuable electronic asset (for example, a phone) in exchange for TOKN, they are demonstrating confidence in the TOKN token as an object with value. It follows that increased confidence in the TOKN token should be sufficient incentive to tokenize assets: **a suitable incentive is a stable token with utility to the agents in the network**.

This directly translates to there being sufficient liquidity for the token and dependable mechanisms cad for its use in purchases and exchange of value. This is achieved by having a liquidity pool with an adequate volume of the TOKN/ALGO token pair.

Furthermore, suitable metrics that can be used to measure demonstrated confidence in the TOKN token are the amount staked in the economy by liquidity providers and the existing amount of asset value tokenized.

3.2 Incentives for asset assurance

A certain fraction of the agents who have assured their asset in each 30-day period is rewarded for doing so. This reward is the incentive.

Assumptions

Agents who assure their asset but are not rewarded feel slightly disincentivized from doing so again.

Dynamics

Only a small fraction of the value of each tokenized asset is set aside for the assurance incentive pool. As illustarted in the `update_assurance_probability_recursive()` function, the likelihood that an agent assures their asset again is proportional to the size of incentive that they get[2]. Since there is only so much incentive to give, there is a tradeoff between the number of agents to reward/ incentivize and the size of incentive to give them. Modeling of this dynamic relationship is ongoing.

```
''' Code snippet '''

fraction_to_reward = 20 # 1 in 20 agents to reward for assuring ownership
.
.
class AgentModel(Agent):

    def __init__(self, unique_id, model):
         self.asset_wealth = np.random.normal(asset_mean, asset_stdev)
         assurance_incentive_pool += 0.025 * self.asset_wealth

    def evaluate_incentive(self, model_assurace_probability, frac):
        # if an agent assured their asset:
        if random.randint(1,10000) in range(1, int(model_assurace_probability*10000)):

            if random.randint(1,frac) == 1: # if rewarded
                self.token_wealth += self.assurance_incentive
                self.update_assurance_probability_recursive(self.assurance_incentive)
            else: # if not rewarded for assurance
                self.assurance_probability *= 0.98 # because assured but was not rewarded

        if self.has_transacted:
            self.token_wealth += self.transaction_incentive

    def update_assurance_probability_recursive(self, incentive):
        self.assurance_probability += (1 - self.assurance_probability) * (1 / (1 +
            math.exp(-1 * (incentive) )))

    def step(self):
        self.evaluate_incentive(self.model.model_assurace_probability, fraction_to_reward)
```

Mathematically, the `update_assurance_probability_recursive()` function is

$$p + (1 - p)\Big(\frac{1}{1 + e^{-v_i/V}}\Big)$$

where v_i is the volume of token value that agent i has in the network and V is the total volume. This function grows the probability of subsequent assurance proportional to the incentive received but never goes beyond one.

Below is the visualization of the results from a test run, showing one agent's asset assurance probability (orange plot) against that of the entire network's.

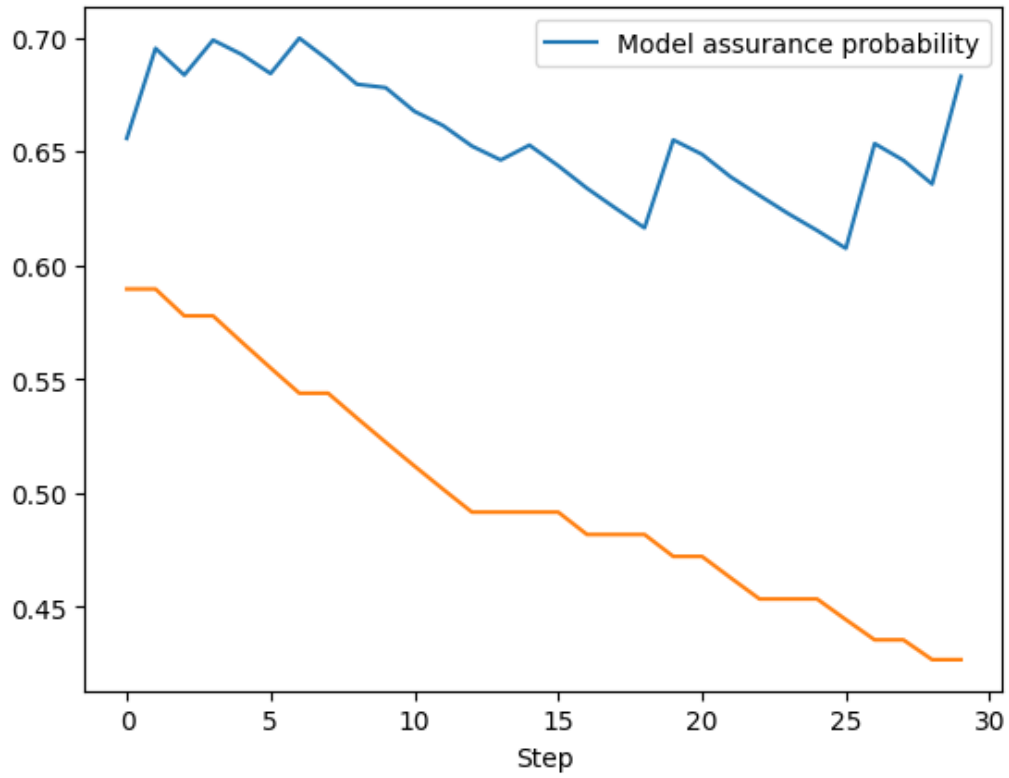

Figure 1: Single agent's assurance probability against that of the network. While the agent's assurance probability has reduced by 0.15 over thirty, 30-day timesteps, that of the model has had an almost 0.05 increase.

3.3 Incentives for provision of liquidity

Liquidity providers are rewarded incentive that is proportional to their stake in the liquidity pool [2].
The dynamics present here are working out the expected [2]impermanent loss and maintaining the expected reward for providing liquidity greater than this.

```
''' Code snippet '''

liquidity_providers_incentive_pool = 0

class LiquidityPoolModel(Model):

    global liquidity_providers_incentive_pool

    def __init__(self, num_liquidity_providers, height, width):
```

[2]This is when the fiat value of a user's deposited tokens reduce over time.

```
        self.TOKN_ALGO_ratio = 4 # TOKN per unit ALGO, given current prices (for 1 TOKN
            backed by $1)
        self.TOKN_volume = 100000 # initialized volumes to provide some liquidity
        self.ALGO_volume = 400000

    def reward_liquidity_providers(self):
        incentive_pool = liquidity_providers_incentive_pool

        for liquidity_provider in self.schedule.agents:
            reward_fraction = liquidity_provider.TOKN_volume / self.TOKN_volume
            reward_value = incentive_pool * reward_fraction
            liquidity_provider.accept_reward(reward_value)

class LiquidityProvider(Agent):

    def __init__(self, unique_id, model):
        self.TOKN_reward = 0

    def accept_reward(self, reward):
        self.TOKN_reward += reward
```

4 Outcomes from simulation

The following variables were captured from the model:

1. **Model assurance probability**. This is the average of the agents' assurance probabilities and is a good metric for the effect of incentivization on the entire network's assurance and hence its ability to demonstrate that TOKN is indeed backed by valuable assets. Therefore, it corresponds to the confidence in the token.
   It is observed that when more agents (1 in lesser n) are rewarded for assuring their asset, the model assurance probability goes up. The [3]dynamics to be modeled here are what fraction to reward is optimal, given the more are rewarded, the less the individual reward and hence the less incentivized an agent feels to assure ownership again.

[3]These dynamics are being worked out in the cadCAD models.

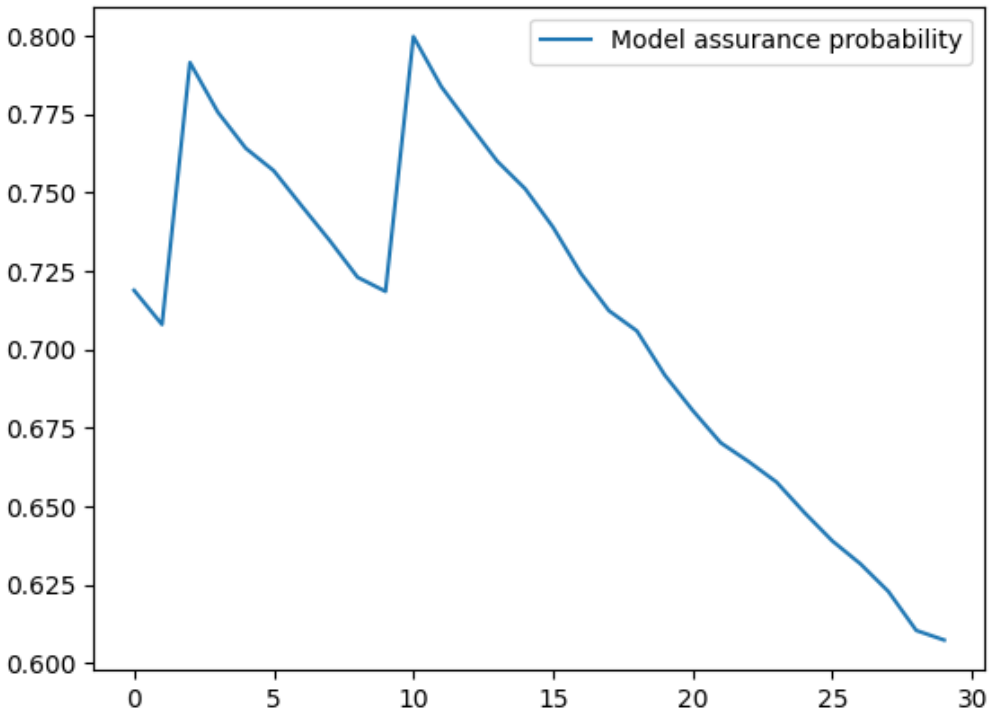

Figure 2: Evolution of model assurance probability.

2. **The protocol revenue**. This it the 30% of tokenized asset value that is retained by the protocol as its revenue. The figure captured is not monthly but rather for the entire time period.

```
Rewarded assured fraction is one in 20 agents who assured asset in last 30-day period.
Number of agents in network is 10
```

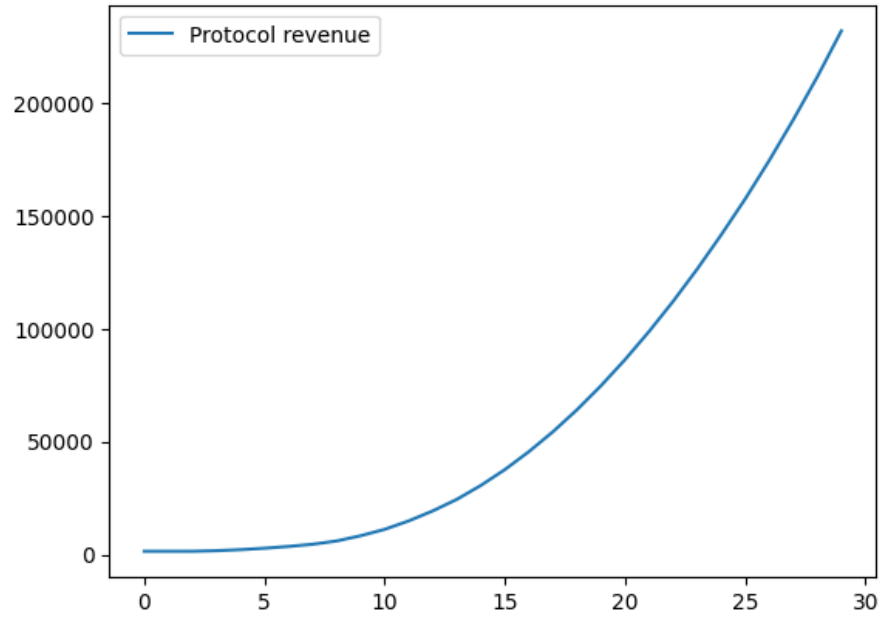

Figure 3: Protocol revenue over 30-day timesteps (economy dynamics yet to be factored in).

5 Entire model code with internal documentation

```
import random, math, numpy as np

from mesa import Agent, Model
from mesa.time import RandomActivation
from mesa.space import SingleGrid, MultiGrid
from mesa.datacollection import DataCollector

# global variables
assurance_incentive_pool = 0
transaction_incentive_pool = 0
economy_token_wealth = 1
asset_mean = 500
asset_stdev = 100
fraction_to_reward = 20
liquidity_providers_incentive_pool = 0

def assurance_incentive_gini(model):
    agent_assurance_incentive = [agent.assurance_incentive for agent in
        model.schedule.agents]
    x = sorted(agent_assurance_incentive)
    N = model.num_agents
    B = sum(xi * (N - i) for i, xi in enumerate(x)) / (N * sum(x))
```

```
    return 1 + (1 / N) - 2 * B

def get_revenue(model):
    return model.protocol_revenue

def get_model_assurance_probability(model):
    return model.model_assurace_probability


class EconomyModel(Model):

    global assurance_incentive_pool, protocol_revenue

    def __init__(self, num_agents, height, width, liquidityPoolModel):
        self.num_agents = num_agents
        self.running = True
        self.grid = MultiGrid(height, width, True)
        self.schedule = RandomActivation(self)
        self.model_assurace_probability = 0
        self.model_assurace_probabilities = list()
        self.protocol_revenue = 0
        self.TOKN_reserve = 0
        self.datacollector = DataCollector(
            model_reporters = {"Model assurance probability":
                get_model_assurance_probability},
            agent_reporters = {"Assurance probability": "assurance_probability"}
        )
        self.myLiquidityPool = liquidityPoolModel

        # create agents
        for agent_index in range(self.num_agents):
            agent = AgentModel(agent_index, self)
            self.schedule.add(agent)
            x = random.randrange(self.grid.width)
            y = random.randrange(self.grid.height)
            try:
                self.grid.place_agent(agent, (x, y))
            except Exception:
                self.grid.place_agent(agent, self.grid.find_empty)

        self.update_model_assurace_probability()

    def update_model_assurace_probability(self):
        sum = 0
        for agent in self.schedule.agents:
            sum += agent.assurance_probability
        self.model_assurace_probability = sum / self.num_agents
        self.model_assurace_probabilities.append(self.model_assurace_probability)

    # grow agent population according to Bass diffusion model, compare different models
    def grow(self):
        pass

    def execute_model(self, n):
        for i in range(n):
```

```
            self.step()
        model_assurace_probabilities = economyModel.datacollector.get_model_vars_dataframe()
        protocol_revenue = economyModel.datacollector.get_model_vars_dataframe()
        model_assurace_probabilities.plot()
        print(f"\nRewarded assured fraction is one in {fraction_to_reward} agents who
            assured asset in last 30-day period.")
        print(f"Number of agents in network is {self.num_agents}. \n")

    def step(self):
        self.datacollector.collect(self)
        self.schedule.step()
        self.update_model_assurace_probability()


class AgentModel(Agent):

    global asset_mean
    global asset_stdev

    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)

        global assurance_incentive_pool, transaction_incentive_pool, economy_token_wealth,
            fraction_to_reward, liquidity_providers_incentive_pool

        self.asset_wealth = np.random.normal(asset_mean, asset_stdev)
        self.model.TOKN_reserve += self.asset_wealth # mint same number of TOKN as price of
            the asset
        self.token_wealth = (1.0-0.3-0.075) * self.asset_wealth
        economy_token_wealth += self.token_wealth
        self.model.protocol_revenue += 0.3 * self.asset_wealth
        assurance_incentive_pool += 0.025 * self.asset_wealth
        transaction_incentive_pool += 0.025 * self.asset_wealth
        liquidity_providers_incentive_pool += 0.025 * self.asset_wealth

        self.has_transacted = False
        self.assurance_incentive = assurance_incentive_pool * (self.token_wealth /
            economy_token_wealth)
        self.transaction_incentive = transaction_incentive_pool * (self.token_wealth /
            economy_token_wealth)

        # we assume agents who have just joined the network are more than indifferent (>50%)
        self.assurance_probability = np.random.normal(0.65, 0.1)
        while (self.assurance_probability < 0 or self.assurance_probability > 1):
            self.assurance_probability = np.random.normal(0.65, 0.1)
        self.model_assurace_probability = self.model.model_assurace_probability

    def tokenize(self, liquidityPoolModel):

        global assurance_incentive_pool, transaction_incentive_pool, economy_token_wealth,
            fraction_to_reward, liquidity_providers_incentive_pool

        asset_value = np.random.normal(asset_mean, asset_stdev)
        TOKN_value = (1.0-0.3-0.075) * self.asset_wealth
        liquidity_confidence_score = TOKN_value / liquidityPoolModel.TOKN_volume
```

```
        if self.assurance_probability * self.model.model_assurace_probability > 0.35 and
            liquidity_confidence_score > 0.01:

            self.asset_wealth += asset_value
            self.model.TOKN_reserve += self.asset_wealth
            self.token_wealth += (1.0-0.3-0.075) * self.asset_wealth
            economy_token_wealth += self.token_wealth
            self.model.protocol_revenue += 0.3 * self.asset_wealth
            assurance_incentive_pool += 0.025 * self.asset_wealth
            transaction_incentive_pool += 0.025 * self.asset_wealth
            liquidity_providers_incentive_pool += 0.025 * self.asset_wealth


    def evaluate_incentive(self, model_assurace_probability, frac):
        if random.randint(1, 10000) in range(1, int(model_assurace_probability*10000)): #
            if assured

            if random.randint(1,frac) == 1: # if rewarded
                self.token_wealth += self.assurance_incentive
                self.update_assurance_probability_recursive(self.assurance_incentive, -14)
            else: # if not rewarded for assurance
                self.assurance_probability *= 0.98 # because assured but was not rewarded.

        if self.has_transacted:
            self.token_wealth += self.transaction_incentive

    def move(self):
        pass

    def transact(self):
        self.has_transacted = True

    def update_assurance_probability_sigmoidal(self, incentive):
        self.assurance_probability = 0.1 + (0.9 / 1 + math.exp(-1 * incentive))

    # using the sigmoid function because reaction is not linear: getting 100 times more
        incentive does not mean an agent is 100 times more likely to assure
    def update_assurance_probability_recursive(self, incentive, ):
        self.assurance_probability += (1 - self.assurance_probability) * (1 / (1 +
            math.exp(-1 * (incentive) )))

    def step(self):
        self.evaluate_incentive(self.model.model_assurace_probability, fraction_to_reward)
        self.tokenize(self.model.myLiquidityPool)


class LiquidityPoolModel(Model):

    global liquidity_providers_incentive_pool

    def __init__(self, num_liquidity_providers, height, width):
        self.num_liquidity_providers = num_liquidity_providers
        self.TOKN_ALGO_ratio = 4 # TOKN per unit ALGO, appproximate given current prices
            (for 1 TOKN backed by $1)
```

```
        self.liquidity_incentive = 1
        self.schedule = RandomActivation(self)
        self.liquidity_provider_TOKN_mean = 100
        self.liquidity_provider_TOKN_variance = 20
        self.TOKN_volume = 100000 # initialized volumes to provide some liquidity
        self.ALGO_volume = 400000

        self.grid = MultiGrid(height, width, True)

        for liquidity_provider_id in range(num_liquidity_providers):
            liquidity_provider = LiquidityProvider(liquidity_provider_id, self)
            self.schedule.add(liquidity_provider)
            x = random.randrange(self.grid.width)
            y = random.randrange(self.grid.height)
            try:
                self.grid.place_agent(liquidity_provider, (x, y))
            except Exception:
                self.grid.place_agent(liquidity_provider, self.grid.find_empty)

    def add_liquidity_providers(self, liquidity_providers_incentive_pool):
        pass

    def take_liquidity(self, TOKN_ALGO_pair):
        if self.model.schedule.steps == 1:
            self.TOKN_volume += TOKN_ALGO_pair[0]
            self.ALGO_volume += TOKN_ALGO_pair[1]
            for agent in self.schedule.agents:
                agent.provide_liquidity()

        for agent in self.schedule.agents:
            agent.add_liquidity()

    def reward_liquidity_providers(self):
        incentive_pool = liquidity_providers_incentive_pool

        for liquidity_provider in self.schedule.agents:
            reward_fraction = liquidity_provider.TOKN_volume / self.TOKN_volume
            reward_value = incentive_pool * reward_fraction
            liquidity_provider.accept_reqard(reward_value)


class LiquidityProvider(Agent):

    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.TOKN_volume = np.random.normal(self.model.liquidity_provider_TOKN_mean,
            self.model.liquidity_provider_TOKN_variance)
        self.ALGO_volume = self.model.TOKN_ALGO_ratio * self.TOKN_volume
        self.TOKN_reward = 0

    def provide_liquidity(self):
        if self.model.schedule.steps == 1:
            self.model.take_liquidity(self.TOKN_volume, self.ALGO_volume)
        else:
            pass
```

```
    def add_liquidity(self):
        pass

    def remove_liquidity(self, fraction):
        pass

    def accept_reward(self, reward):
        self.TOKN_reward += reward

    def withdraw_from_pool(self):
        self.remove_liquidity(1)

    def step(self):
        self.provide_liquidity()


if __name__ == "__main__":

    liquidityPoolModel = LiquidityPoolModel(50, 10, 10)
    economyModel = EconomyModel(4, 10, 10, liquidityPoolModel)

    economyModel.execute_model(30)

    agent_assurance_probabilities = economyModel.datacollector.get_agent_vars_dataframe()
    print(agent_assurance_probabilities.head(n=40))
    agent_assurance_probabilities.plot()
```

6 Ongoing work

cadCAD (complex adaptive dynamics Computer-Aided Design) is a python based modeling framework for research, validation, and Computer Aided Design of complex systems [1]. This tool is being used to model dynamics in the economy that extend past the purview of an agent-based model, as the one herein.

References

[1] cadCAD.org. What is cadcad.

[2] Philipp Hülsemann and Andranik Tumasjan. Walk this Way! Incentive Structures of Different Token Designs for Blockchain-Based Applications. *International Conference on Information Systems (ICIS) Proceedings*, 2019.